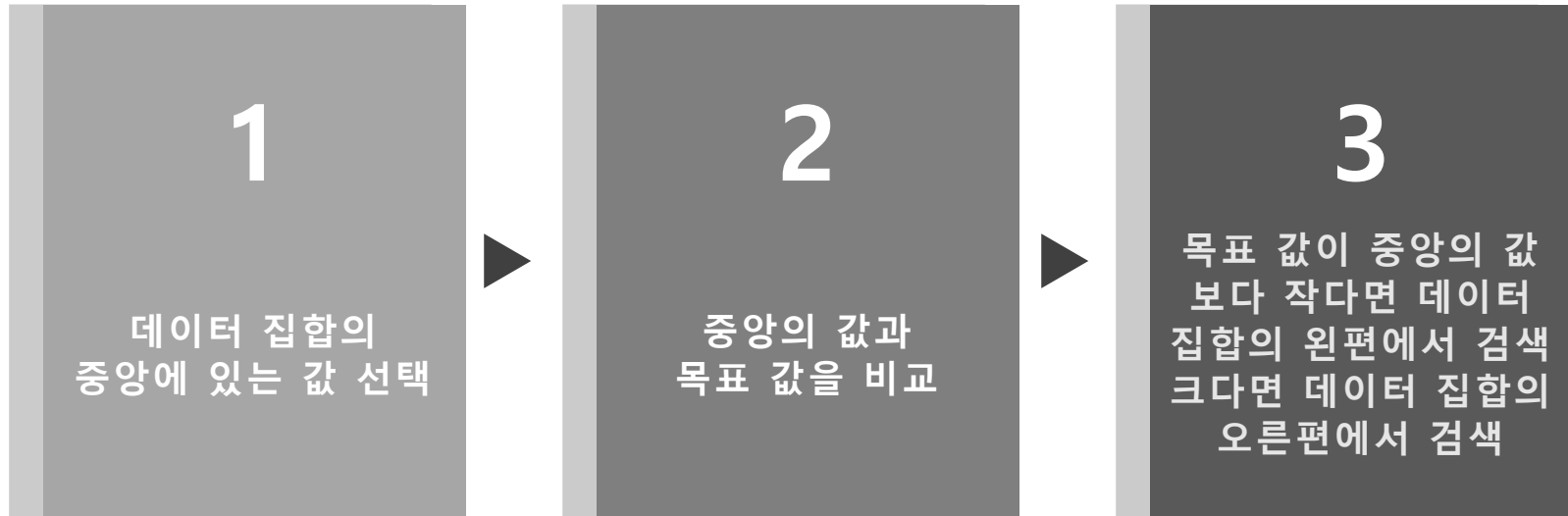


이진 탐색 트리





- ◆ 정렬된 배열에서 사용할 수 있는 탐색 알고리즘
- ◆ 값을 찾을 때까지 1번~3번 과정을 반복



이진 탐색의 예 (e.g., 67)

0	1	2	3	4	5	6	7	8	9	10
1	7	11	12	14	23	67	139	672	871	912

$$(0 + 10)/2$$

1	7	11	12	14	23	67	139	672	871	912
---	---	----	----	----	----	----	-----	-----	-----	-----

$$(6 + 10)/2$$

1	7	11	12	14	23	67	139	672	871	912
---	---	----	----	----	----	----	-----	-----	-----	-----

$$(6 + 7)/2$$

1	7	11	12	14	23	67	139	672	871	912
---	---	----	----	----	----	----	-----	-----	-----	-----

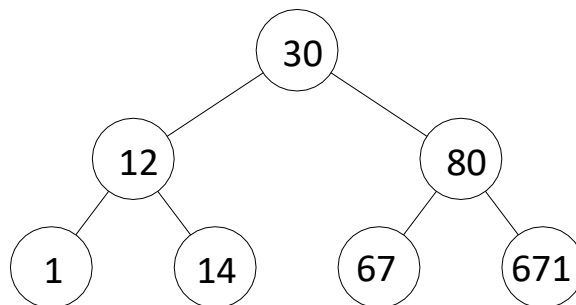
Time: $O(\log n)$



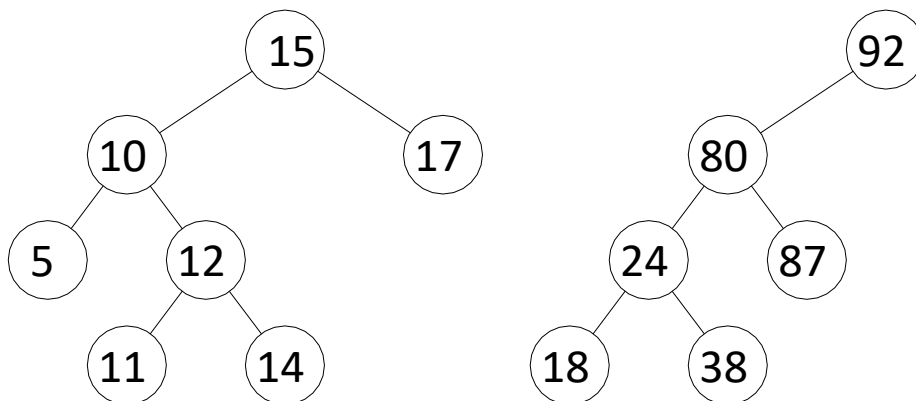


1 각 노드가 다음 규칙을 따르는 이진 트리

- 1 왼쪽 자식 노드는 부모보다 작고, 오른쪽 자식 노드는 부모보다 크다



2 이진 탐색 트리의 예





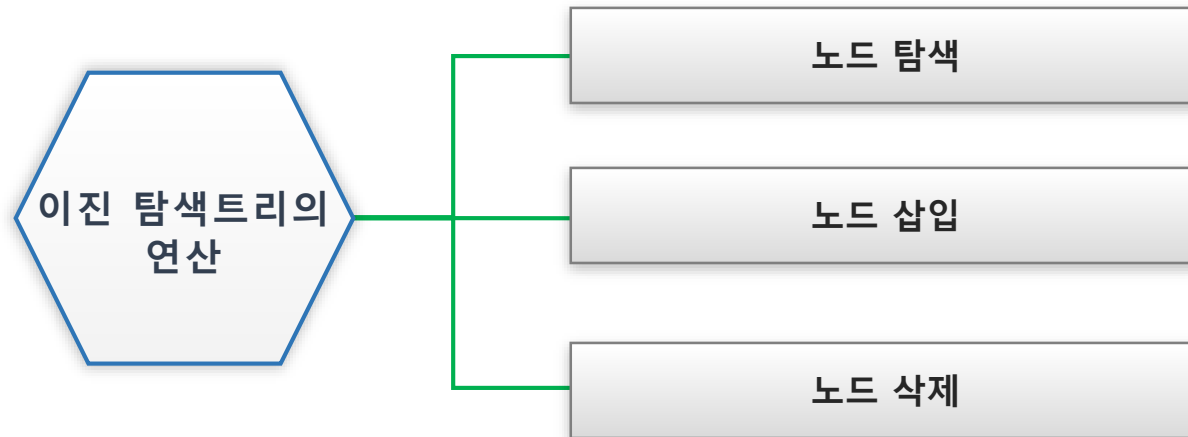
1 이진 탐색 vs 이진 탐색 트리

- 같은 원리에 의한 탐색구조
- 이진 탐색: 배열에 정렬되어 저장된 데이터를 탐색하는 알고리즘
- 이진 탐색 트리: 각 내부노드 v 가 $\text{key}(v.\text{left}) \leq \text{key}(v) \leq \text{key}(v.\text{right})$ 을
만족하는 이진 트리(자료구조)
 - 이진 탐색과 유사하게 탐색 가능

2 이진 탐색 트리에서의 시간 복잡도

- 균형트리: $O(\log n)$
- 불균형 트리: $O(n)$, 순차탐색과 동일

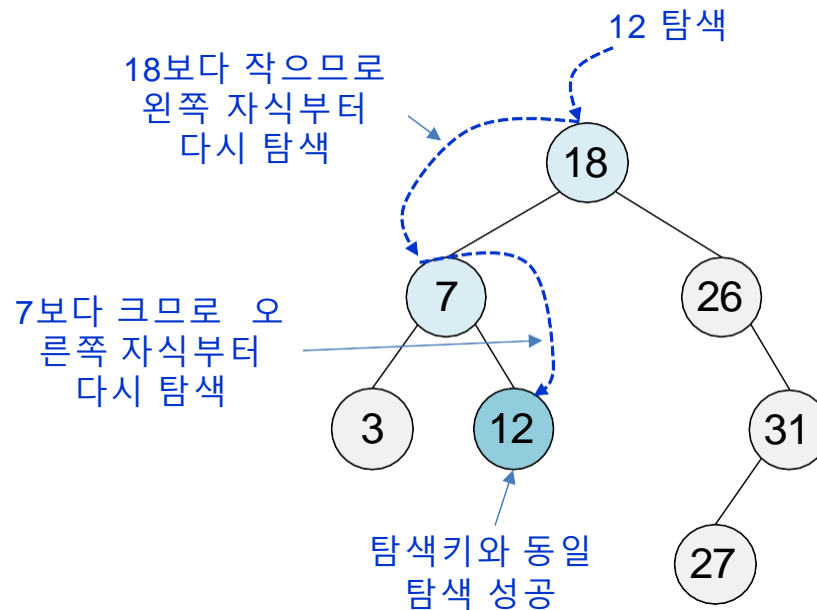






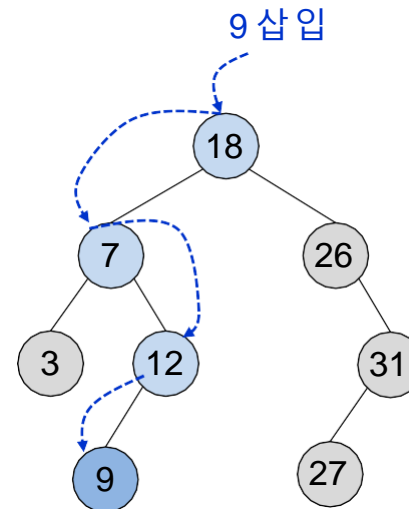
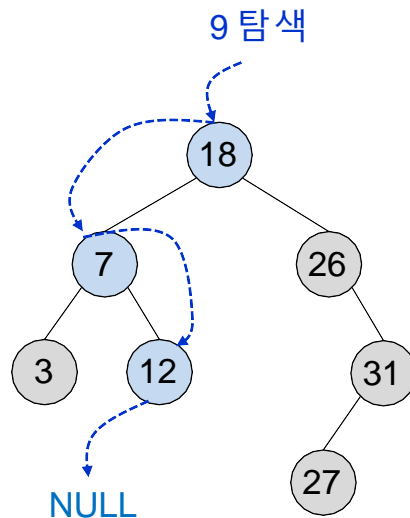
- 탐색 연산

- 비교한 결과가 같으면 탐색이 성공적으로 끝난다.
- 키 값이 루트보다 작으면 왼쪽 자식을 기준으로 다시 탐색
- 키 값이 루트보다 크면 오른쪽 자식을 기준으로 다시 탐색





- 삽입 연산
 - 먼저 탐색을 수행
 - 탐색에 실패한 위치에 새로운 노드를 삽입





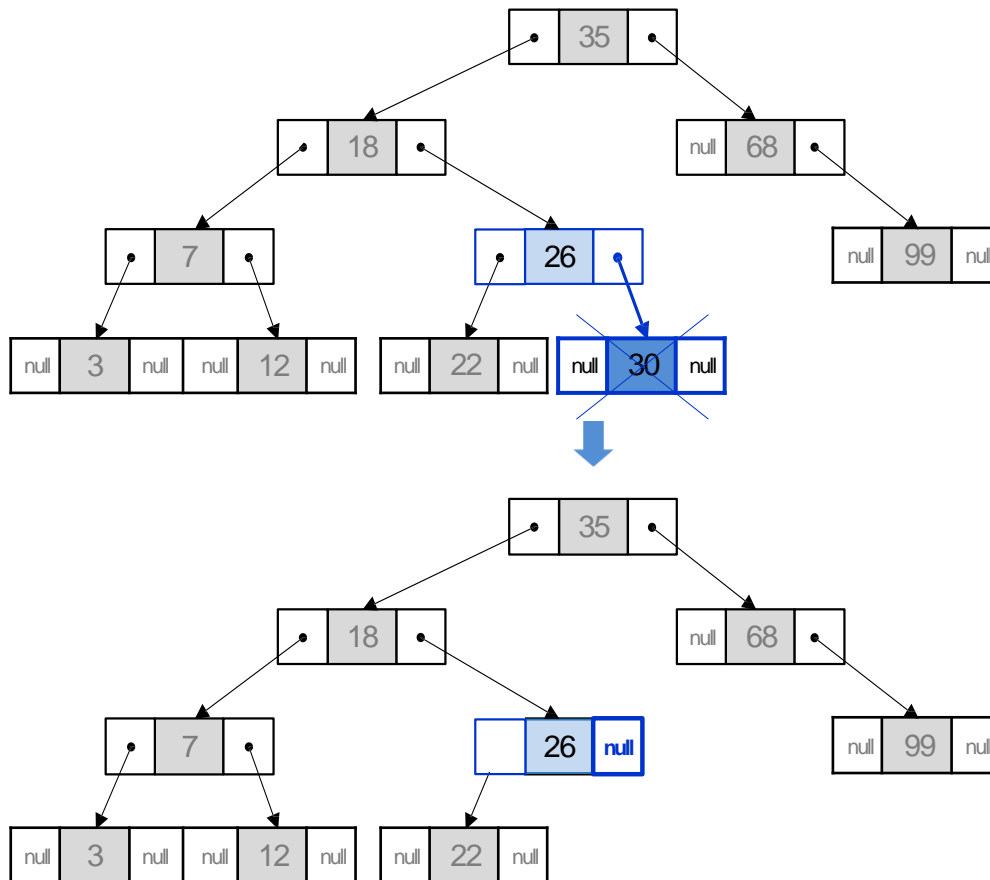
- 삭제 연산
 - 3가지 경우 존재
 1. 삭제하려는 노드가 단말 노드일 경우
 2. 삭제하려는 노드가 왼쪽이나 오른쪽 서브 트리 중 하나만 가지고 있는 경우
 3. 삭제하려는 노드가 두 개의 서브 트리 모두 가지고 있는 경우





Case 1: 단말 노드 삭제

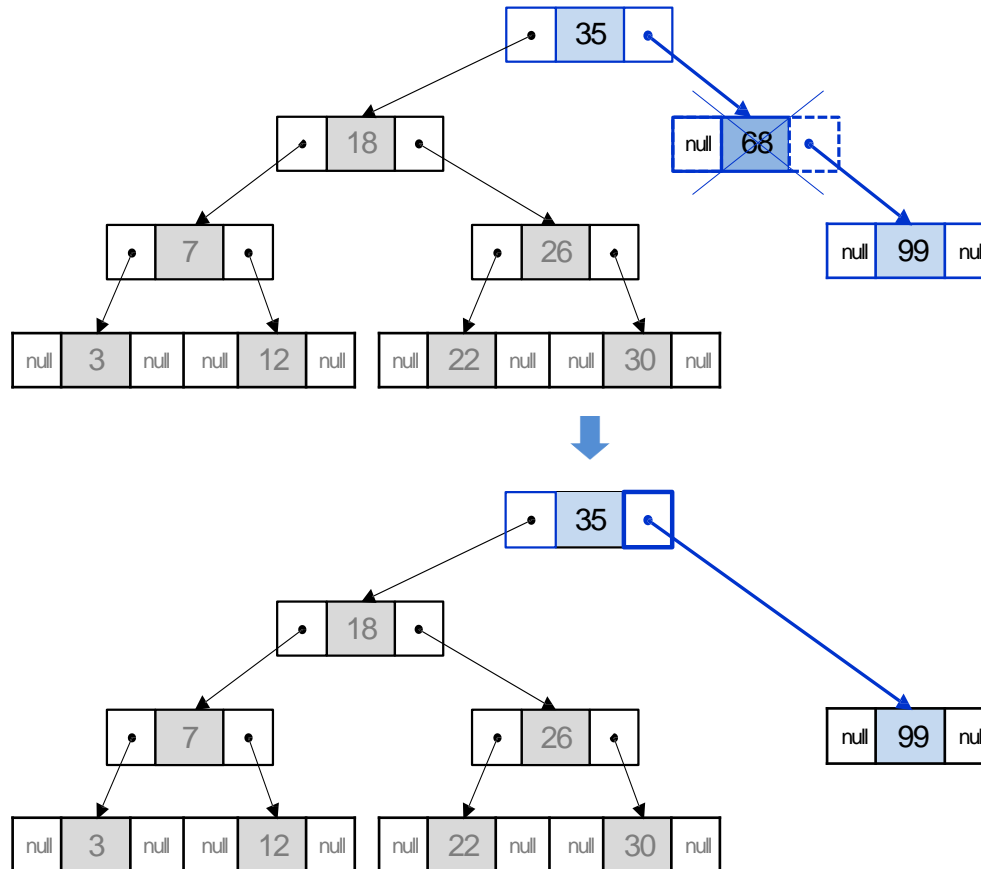
- 단말 노드의 부모 노드를 찾아서 링크 제거





Case 2: 자식이 하나인 노드 삭제

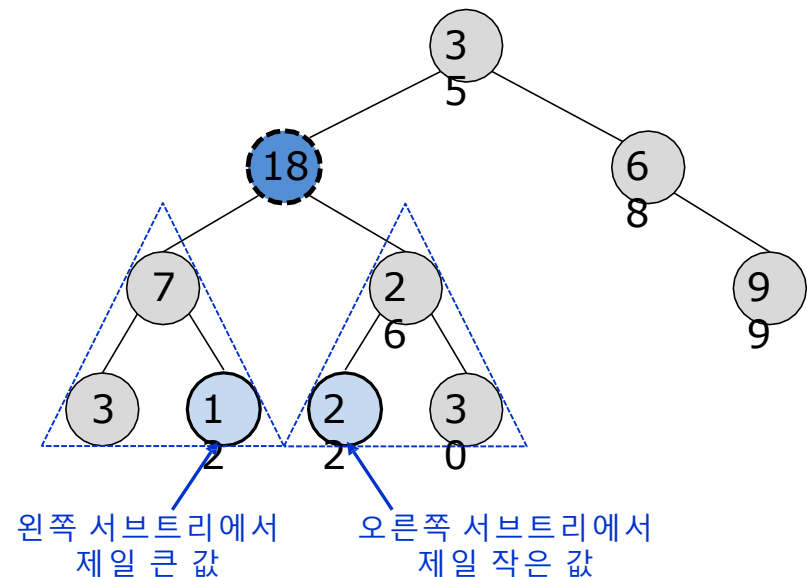
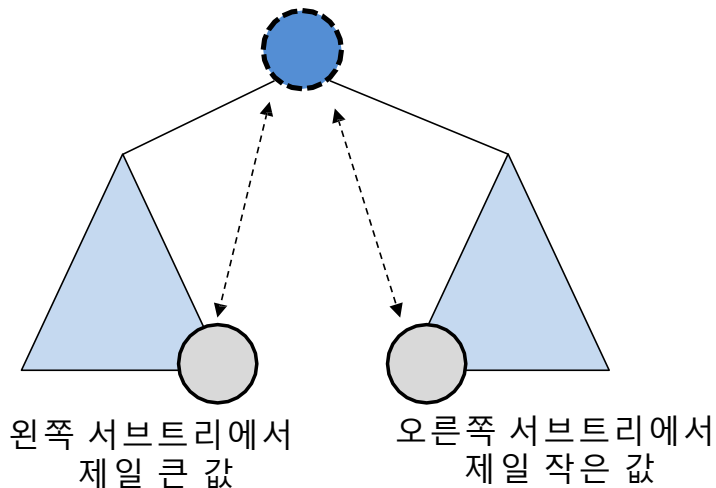
- 노드를 삭제하고 서브 트리를 삭제된 노드의 부모 노드에 연결





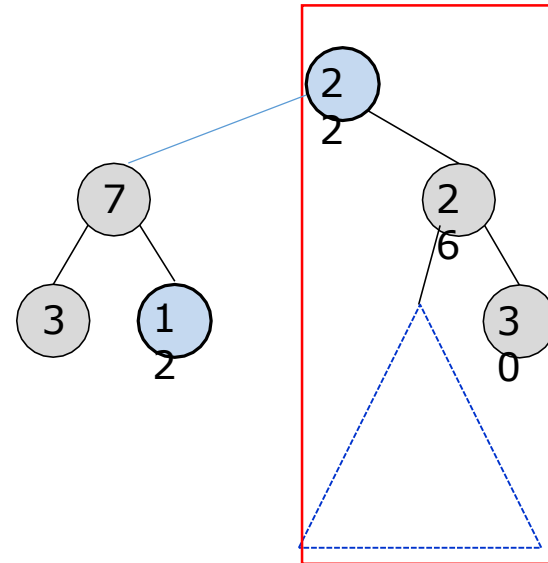
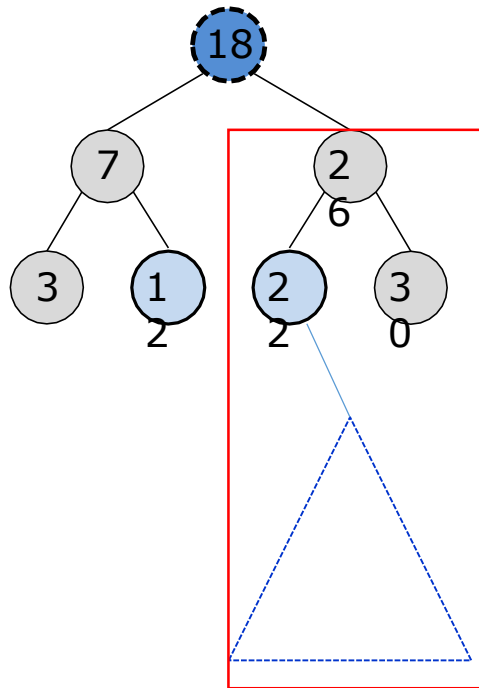
Case 3: 두 개의 자식을 가진 노드 삭제

- 가장 가까운 값을 가진 노드(왼쪽 서브 트리에서 제일 큰 값 OR 오른쪽 서브 트리에서 제일 작은 값)를 삭제 노드 위치로 이동
- 가장 가까운 값을 가진 노드는 자식이 최대 1개임이 보장되므로, 이동된 위치에 case 1 또는 case 2를 적용



이진 탐색 트리

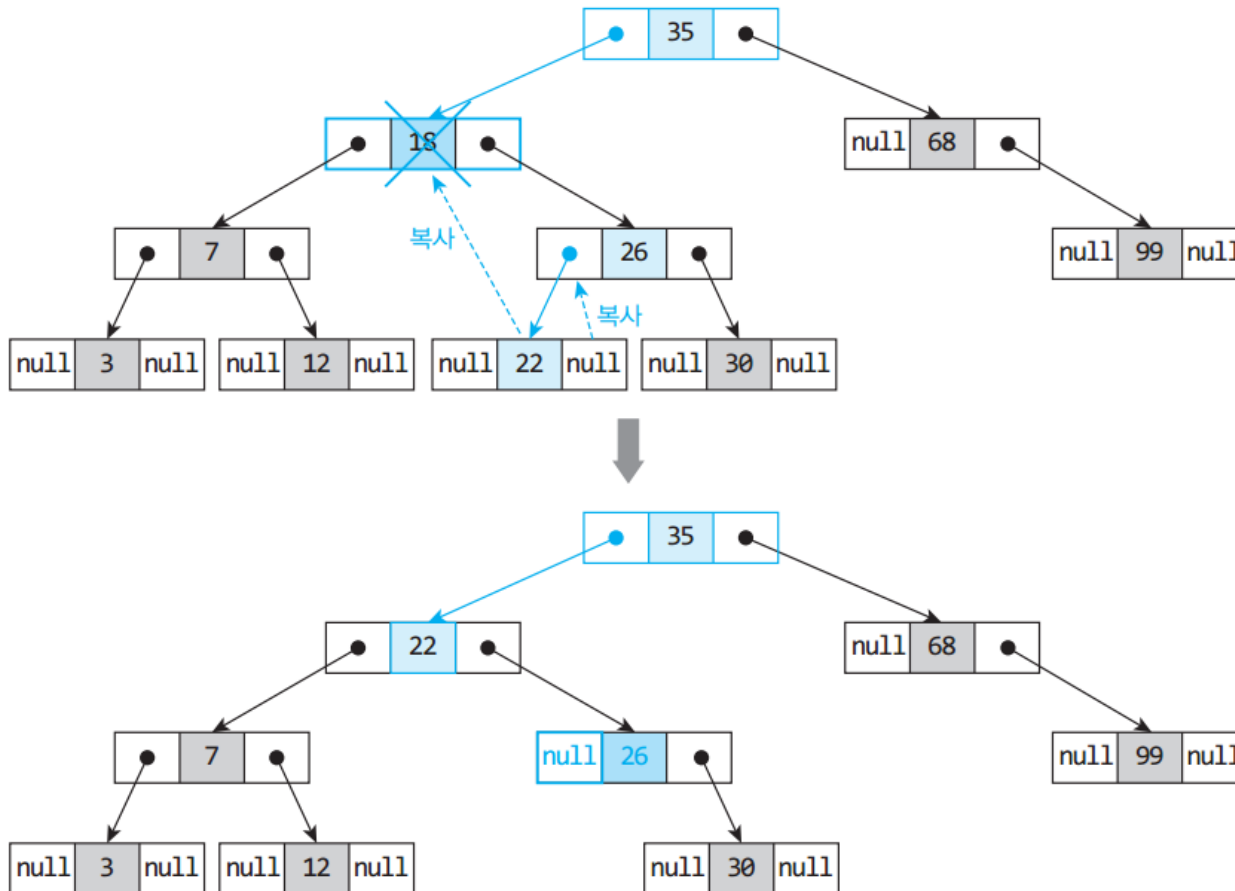
COMPUTER ENGINEERING





Case 3: 두 개의 자식을 가진 노드 삭제

- 노드 18의 삭제 과정





이진 탐색 트리의 성능

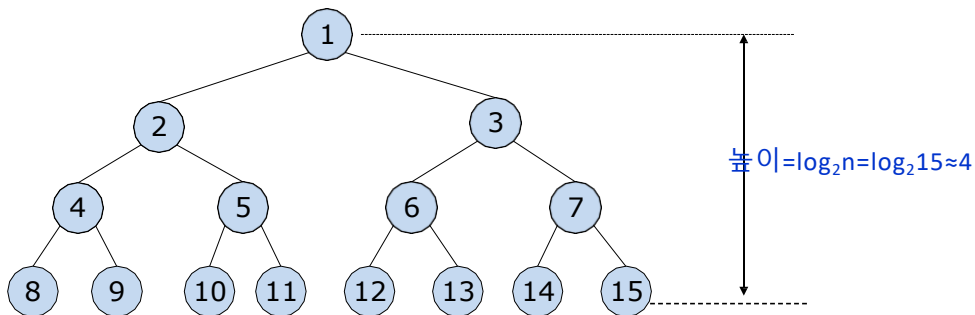
- 이진 탐색 트리에서의 탐색, 삽입, 삭제 연산의 시간 복잡도는 트리의 높이를 h 라고 했을 때, h 에 비례한다

□ 최선의 경우

- 이진 트리가 균형적으로 생성되어 있는 경우

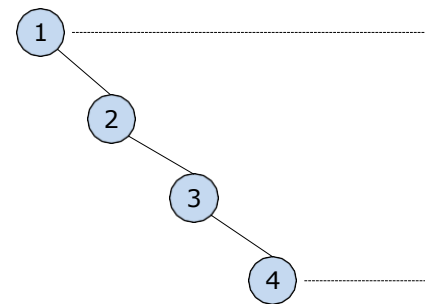
$$h = \log_2 n$$

- 시간복잡도: $O(\log n)$



□ 최악의 경우

- 경사 이진트리: $h = n$
- 시간복잡도: $O(n)$





```
class Node {
public:
    int data;
    Node* parrent;
    Node* rChild;
    Node* lChild;

    // 생성자
    Node() {
        this->data = NULL;
        this->parrent = NULL;
        this->rChild = NULL;
        this->lChild = NULL;
    }

    Node(int data) {
        this->data = data;
        this->parrent = NULL;
        this->rChild = NULL;
        this->lChild = NULL;
    }

    // 소멸자
    ~Node() {
    }

    // 왼쪽 자식에 추가
    void insertlChild(Node* lChild) {
        this->lChild = lChild;
        lChild->parrent = this;
    }

    // 오른쪽 자식에 추가
    void insertrChild(Node* rChild) {
        this->rChild = rChild;
        rChild->parrent = this;
    }

    // 해당 노드의 자식 수 출력
    void printDegree() { ... }
    // 해당 노드의 깊이 출력
    void printDepth() { ... }
};
```





```
class BST {
public:
    Node* root;           // root 노드
    int height;           // 트리의 높이
    // 생성자
    BST() {
        root = NULL;
        height = 0;
    }
    // 소멸자
    ~BST() {
        this->treeDestructor(this->root);
    }
    // 노드 삽입 연산 (트리의 높이를 함께 고려)
    void insertNode(int data) {
        int height = 0;
        if (root == NULL) { ... }
        else { ... }
    }
    // 노드 탐색 연산
    Node* findNode(int data) {
        Node* tmp = root;
        while (tmp != NULL) {
            if (tmp->data == data)
                return tmp;
            else {
                if (tmp->data < data)
                    tmp = tmp->rChild;
                else
                    tmp = tmp->lChild;
            }
        }
        return NULL;
    }
};
```



```
// 트리의 높이 출력
void printHeight() { ... }
// 노드 삭제 연산
void deleteNode(int data) { ... }
// node의 오른쪽 자식에서 최솟값 탐색
Node* findMinimum(Node* node) { ... }
// 전위 순회(pre-order traversal) 결과 출력
void printPreorderTraversal(Node* root) { ... }
// 후위 순회(post-order traversal)하며 트리의 모든 노드 삭제 (소멸자에서 사용)
void treeDestructor(Node* root) {
    if (root == NULL)
        return;

    if (root->lChild != NULL)
        this->treeDestructor(root->lChild);
    if (root->rChild != NULL)
        this->treeDestructor(root->rChild);
    delete(root);
}
};
```

